Comparison of Two Activity Analyses for Automatic Differentiation: Context-Sensitive Flow-Insensitive vs. Context-Insensitive Flow-Sensitive

Jaewook Shin and Paul D. Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Ave. Argonne, IL 60439
{jaewook,hovland}@mcs.anl.gov

Abstract

Automatic differentiation (AD) is a family of techniques to generate derivative code from a mathematical model expressed in a programming language. AD computes partial derivatives for each operation in the input code and combines them to produce the desired derivative by applying the chain rule. Activity analysis is a compiler analysis used to find active variables in automatic differentiation. By lifting the burden of computing partial derivatives for passive variables, activity analysis can reduce the memory requirement and run time of the generated derivative code. This paper compares a new context-sensitive flow-insensitive (CSFI) activity analysis with an existing context-insensitive flow-sensitive (CIFS) activity analysis in terms of execution time and the quality of the analysis results. Our experiments with eight benchmarks show that the new CSFI activity analysis runs up to 583 times faster and overestimates up to 18.5 times fewer active variables than does the existing CIFS activity analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors - Compilers; G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation - Automatic Differentiation

Keywords

automatic differentiation, activity analysis

1. Introduction

Automatic differentiation (AD) automatically performs the differentiation of a given mathematical model expressed in a programming language. Since it retains the accuracy of the original model and is easy to apply, AD has been widely used in scientific and engineering applications for the past decade [8, 4]. For example, AD can be used in physics modeling to study sensitivity of the model's behavior with respect to one of the input variables. Another class of applications of AD is design optimization for an engineering object such as aircraft.

subroutine head(x,y) double ...,dimension(2)::x double ...,dimension(1)::y	subroutine head(x,y) type(active):: x(1:2) type(active):: y(1:1)
c$openad INDEPENDENT(x) y(1)=x(1)*x(2) c$openad DEPENDENT(y) end subroutine	y(1)%v = x(1)%v * x(2)%v y(1)%d = x(1)%d * x(2)%v y(1)%d += x(2)%d * x(1)%v end subroutine
(a) Input	(b) Output

Figure 1. An example of automatic differentiation.

From AD's viewpoint, an input program is a sequence of differentiable operations with a set of input and output variables. In order to compute the derivative of an output variable with respect to an input variable, AD computes a partial derivative for each operation and applies the *chain rule* of calculus to combine them into a final derivative. Computing derivatives for all output variables with respect to all input variables requires partial derivatives to be computed for all intermediate variables with respect to all input variables, thus requiring as much memory space and compute cycles. Figure 1(a) shows a simple Fortran code example to illustrate how automatic differentiation works. In this example, we declare `x` as *independent* and `y` as *dependent* using compiler directives. In the output shown in (b), note that the types of `x` and `y` are modified to `active` type, which is a structure of two variables: `v` to represent the original value and `d` to represent the derivative. The first statement computes the original function as in (a), and the other two statements compute the gradient of y $grad(y) = \frac{dy}{dx(1)} \times dx(1) + \frac{dy}{dx(2)} \times dx(2)$.

Often, however, scientists are interested in the derivatives of a subset of the output variables with respect to a subset of the input variables. We call such input variables of interest *independent variables* and the output variables of interest *dependent variables*. Users provide the independent and dependent variable information to AD tools by specifying them, for example, as compiler direc-

SAC'07 March 11-15, 2007, Seoul, Korea

tives. We say a variable is *varied* if it (transitively) depends on any independent variable and *useful* if any dependent variable (transitively) depends on it. A variable is *active* if it is both varied and useful at a program point. When the number of the specified independent variables is less than the input variables or the number of the specified dependent variables is less than the output variables, some intermediate variables may be either not *varied* by any independent variable or not *useful* in computing any dependent variable. Such variables are said to be *passive*, or *inactive*. Partial derivatives for passive variables need not be computed because, even though they are computed, the values will be zeros or will not contribute to the desired derivatives. By not computing partial derivatives for such intermediate variables, both the memory requirement and the run time of the generated program can be reduced. *Activity analysis* finds active intermediate variables for which partial derivatives need to be computed. Activity analysis is *flow-sensitive* if it takes into account the order of statements and the control flow structure of procedures, and *context-sensitive* if it is an interprocedural analysis that considers only realizable call-return paths.

Activity analysis has been known for some time [5], and several implementations are used in AD tools. Until recently, we have been using an implementation of an activity analysis algorithm, which is based on an iterative dataflow analysis framework. Since the implementation uses an interprocedural control flow graph (ICFG), we call this implementation *ICFGAA*. The dataflow analysis framework is simple and clean, but for activity analysis it does not scale well when the input size is large. Also, the context-insensitive nature of the algorithm undermines the quality of the output by overestimating active variables. In Section 2, we elaborate on these aspects of ICFGAA.

We have therefore developed a new activity analysis algorithm, called *variable dependence graph* activity analysis (VDGAA). The key idea of VDGAA is intuitive. First, we generate a variable dependence graph [12] to represent data flow between variables. An edge is generated from node A to node B if there is a value flow from variable A to B. Next, the nodes are colored in two sweeps, forward, starting from the independent variable nodes while coloring the visited nodes with one color, and backward, starting from the dependent variable nodes using another color. All variables that are colored by both colors are selected as active. To support context sensitivity, we maintain a call stack while navigating the graph (Section 3). We have implemented the new activity analysis algorithm and used it on a set of eight benchmarks, including the large MIT General Circulation Model (GCM) [1]. From our experiments, we observe that VDGAA runs up to 583 times faster and improves the quality of the output by up to 18.5 times over ICFGAA. In other words, VDGAA overestimates significantly fewer active variables, thereby reducing the memory requirement and the run time of the derivative code.

Our contributions in this paper are as follows:

- A new CSFI activity analysis algorithm.
- Comparison of the new CSFI activity analysis with the existing CIFS activity analysis in terms of run time and output quality.
- Implementation and experimental evaluation of the new algorithm on eight benchmarks.

The remainder of this paper is organized as follows. In the next section, we describe the existing CIFS activity analysis and use examples to motivate our research. In Section 3, the new CSFI activity analysis algorithm is described. In Section 4, we present our implementation and experimental results. In Section 5, we discuss related research. We conclude in Section 6.

2. Background

In this section, we motivate our research by explaining how the existing CIFS activity analysis (ICFGAA) works. We then discuss how its run time can be increased and how it can overestimate active variables.

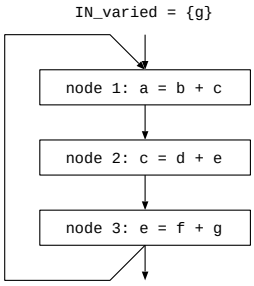

Figure 2. When implemented in iterative dataflow analysis framework, activity analysis can iterate more than what is limited by the depth of the control flow graph.

Table 1. OUT sets of the nodes in Figure 2 in iterative *varied* analysis

Iterations	Node 1	Node 2	Node 3
1	{g}	{g}	{e, g}
2	{e, g}	{c, e, g}	{c, e, g}
3	{a, c, e, g}	{a, c, e, g}	{a, c, e, g}

ICFGAA starts by building an interprocedural control flow graph, which is generated by connecting the control flow graphs of the procedures in a given program. For each call site in procedure A to procedure B, the basic block BB containing the call is split into two, BB1 and BB2. BB1 is from the beginning of BB up to the call statement, and BB2 is from the statement right below the call to the end of BB. A call edge from BB1 to the entry node of procedure B and a return edge from the exit node of procedure B to BB2 are inserted. Once the ICFG is complete, the two iterative dataflow analyses follow. First, *useful* variables are propagated backward along the ICFG edges starting from the exit node of the program. Next, *varied* variables are obtained by propagating forward starting from the entry node of the program.

A program analysis is *separable* if the solution values are not coupled with each other [16, 10]. For example, reaching definition analysis and live variable analysis are separable because, for reaching definition analysis, the reachability of one definition at a certain program point does not affect the reachability of other definitions at the same program point. Several program analyses are not separable, however, such as reaching constant and points-to analysis. So is activity analysis. The number of iterations in separable dataflow analyses is bounded by the *depth* of the control flow graph [2]. In the case of activity analysis, however, the

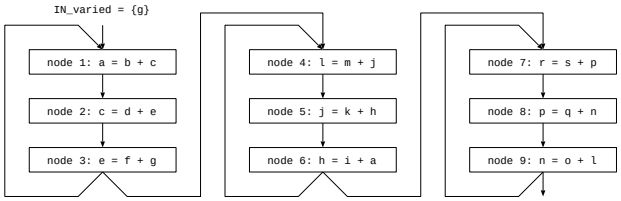

Figure 3. Concatenated, short backward dependence chains.

upper bound can be much higher than just the depth of the program's control flow graph. Figure 2 is an example code snippet to show this point. Since the depth of the graph in the example is 1, the upper bound on the number of iterations is 3 (depth + 2) if the analysis is separable: once to propagate the information down, once to propagate the lower block information along the back edge and, once to check that there are no further changes. If we are doing a *varied* analysis, which propagates the variables varied by the independent variables, and if the IN set of node 1 is $\{g\}$, the analysis needs to iterate four times (Table 1), which is higher than the upper bounds on separable analysis cases. In this example, we deliberately arranged the statements so that there is a dependence chain in the reverse textual order: node $3 \rightarrow 2 \rightarrow 1$. Although we have shown the smallest example, the number of iterations can be arbitrarily large as the length of the backward dependence chain increases. In our experiments with the MIT GCM code, the varied analysis of ICFGAA iterated 31 times. While it is hard to imagine any backward dependence chain as long as 30, short backward dependence chains can be concatenated via forward dependencies. In Figure 3, the three backward dependence chains of length 3 are concatenated and have the same effect as one backward dependence chain of length 7.

```
    subroutine head(x,y,a,b)
    double precision :: x,y,a,b
c$openad INDEPENDENT(x)
        call foo(x, b)
        call foo(a, y)
c$openad DEPENDENT(y)
    end subroutine
```

```
subroutine foo(f,g)
    double precision :: f,g
    g = f
end subroutine
```

Figure 4. An example to show how ICFGAA overestimates active variables.

ICFGAA is context insensitive in that dataflow analysis values can flow along unrealizable control paths. In other words, values can flow into a procedure along the call edge of one call site and flow out of the procedure along the return edge of another call site. Figure 4 is a small example to illustrate this point. In subroutine `head`, `x` is declared as an independent variable and `y` as a dependent variable by using compiler directives. The ICFG of this code is shown in Figure 5. For each call to `foo`, two nodes are created, one call node representing the call statement before call and one return node for the call statement after the call. Node 6 and 7 are the call and return nodes for the first call statement, and likewise node 10 and 11 are for the second call. For procedure `foo`, three nodes are generated. Node 8 and 9 represent the entry and exit node of the procedure, and node 14 is for the statement within it. All other irrelevant nodes are abbreviated for simplicity. Table 2(a)

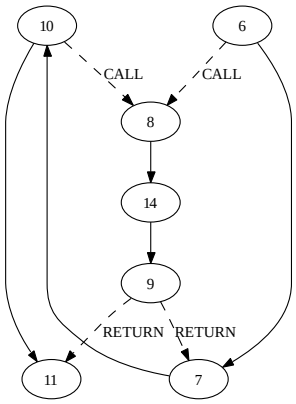

Figure 5. The ICFG of the code in Figure 4.

shows how OUT sets of the nodes change over iterations in the *varied* analysis. Along call and return edges, parameter binding information is used to replace an actual parameter with the corresponding formal parameter and vice versa. Table 2(b) shows how IN sets of the nodes are changed in the *useful* analysis. Value propagation in the *useful* analysis is similar to that of the *varied* analysis except that it propagates backward along the edges. Also, note that variable `g` is not useful anymore above the assignment in procedure `foo`. According to ICFGAA, `x` is active because it is both varied and useful in the IN set of node 6, and `y` is active because of the IN set of node 11. Further, `f` and `g` are active because of the IN sets of node 14 and node 9, respectively. In the code, however, there is no value flow from the independent variable `x` to the dependent variable `y`. Hence, no variables are active.

Table 2. Iterative dataflow analysis values for the nodes of Figure 5

Order	Node	Iter 1	Iter 2
1	6	{x}	{x}
2	7	{x}	{x,b}
3	10	{x}	{x,b}
4	8	{f}	{f}
5	14	{f}	{f}
6	9	{f,g}	{f,g}
7	11	{x,a,y}	{x,a,y,b}

(a) *Varied* analysis (OUT sets)

Order	Node	Iter 1	Iter 2
1	11	{y}	{y}
2	10	{y}	{y,a}
3	7	{y}	{y,a}
4	9	{g}	{g}
5	14	{f}	{f}
6	8	{f}	{f}
7	6	{x,y}	{x,y,a}

(b) *Useful* analysis (IN sets)

In this section, we have shown that ICFGAA can iterate a large number of times when the order of the statements in a dependence chain is opposite to the propagation direction of the analysis. Also, we have shown how ICFGAA overestimates active variables because of context insensitivity. Based on this observation, we developed a new activity analysis algorithm that is context sensitive

but does not use an iterative dataflow framework.

3. Algorithm

In this section, we describe the new activity analysis algorithm VDGAA. The algorithm consists of three major steps, as follows.

1. Build a variable dependence graph (VDG).
2. Propagate forward from the independent variable nodes in the VDG to find *varied* variable nodes.
3. Propagate backward from the dependent variable nodes in the VDG to find *useful* variable nodes.

```
Build-VDG(program PROG)
  VDG ← new Graph
  DepMatrix ← new Matrix
  for each procedure P ∈ CallGraph(PROG) in postorder
    for each statement Stmt ∈ P
      for each (Src,Dst) pair ∈ Stmt
        VDG.addEdge(Src → Dst, FLOW, P)
        DepMatrix[P][Src][Dst] = true
      if (Stmt has procedure calls)
        for each (Actual,Formal) pair of a Call ∈ Stmt
          VDG.addEdge(Actual → Formal, CALL, P)
          if (Formal is a reference parameter)
            VDG.addEdge(Formal → Actual, RETURN, P)
          Label the two edges with the address of Call

    DepMatrix[P] ← TransitiveClosure(DepMatrix[P])
    for each formal parameter pair (F1,F2) of P
      if (DepMatrix[P][F1][F2])
        VDG.addEdge(F1 → F2, PARAM, P)
        for each call to P from P'
          DepMatrix[P'][Actual(F1)][Actual(F2)] = true
```

Figure 6. Algorithm: Build variable dependence graph.

A variable dependence graph (VDG) is a tuple (V, E), where a node N ∈ V represents a variable in a program uniquely and an edge (n1, n2) ∈ E represents a data dependency from node n1 to n2 [12]. Since the multiple definitions of a variable are all mapped to a node, the information for flow sensitivity is lost when the graph is built. Figure 6 shows an algorithm to build VDG. For each statement in a given program, we generate a FLOW edge from each source variable to the destination variable. If a statement contains procedure calls, we also add a CALL edge from each actual parameter to the corresponding formal parameter and a RETURN edge from the formal parameter to the actual parameter if it is a reference parameter. These edges represent value flow between pairs of variable nodes. In addition to FLOW, CALL, and RETURN, there is one more edge type, PARAM, which summarizes data flow among formal parameters. We add a PARAM edge from formal parameter f1 to f2 whenever there is a path from f1 to f2. The PARAM edges are added to allow multiple traversals along the same edge when there are multiple call sites for a procedure. To insert these PARAM edges, we set an element of the procedure's dependence matrix to true whenever a FLOW edge is created. After building the VDG of a procedure, we apply Warshall's *transitive closure* algorithm to add PARAM edges for pairs of formal parameter nodes whenever there is a path from one to the other.

The *varied* analysis shown in Figure 7 propagates from the independent variable nodes forward along the VDG edges. To support context sensitivity, we use a call stack. When a CALL edge is followed, we push the call site label of the edge onto the stack, and we pop the stack top when the corresponding RETURN edge is followed or the propagation retreats back along the CALL edge. Before a RETURN edge is followed, we compare the current top of the stack with the edge label. The edge is followed only when they match. One exception is when the top of the stack keeps a special value called VTG (for Value Through Globals), in which case we allow any RETURN edges to be followed. Even when the stack top is VTG, however, CALL edges can be followed, maintaining the stack normally. Note that we still follow the realizable value flow paths even when VTG is used.

The *useful* analysis is similar to the varied analysis. However, it traverses the VDG backward starting from the dependent variable nodes. Another difference is that we do not visit a node unless it is already marked as *varied*. Finally, when a node is marked as useful, we mark the variable *active* as well.

```
Node::Mark_Varied_Node(stack CALLS, procedure CurrProc)
  Mark this as varied
  for each successor node N and the edge E
    if (N.onPath ∨ E.visited) continue
    N.onPath ← true
    switch (E.kind)
      case CALL:
        CALLS.push(E.label);E.visited ← true
        N.markVaried(CALLS,E.sinkProc)
        CALLS.pop();break
      case RETURN: // VTG: Value Through Globals
        if (CALLS.top() == E.label ∨ CALLS.top == VTG)
          if (CALLS.top != VTG) CALLS.pop()
          E.visited ← true
          N.markVaried(CALLS,E.sinkProc)
          if (CALLS.top != VTG) CALLS.push(E.label)
        break
      default :
        if (E.kind != PARAM) E.visited ← true
        if (CurrProc != E.proc)
          CALLS.push(VTG)
          N.markVaried(CALLS,E.proc)
          CALLS.pop()
        else
          N.markVaried(CALLS,E.proc)
    N.onPath ← false
```

Figure 7. Algorithm: Mark *varied* nodes.

Figure 8 is the VDG generated for the code in Figure 4. We use this VDG to show how VDGAA works. All variables in the input code are mapped to their own nodes in the graph. A FLOW edge from F to G is generated because of the statement in procedure `foo`, and a PARAM edge between the same pair of nodes is generated as a result of the transitive closure operation. All other edges are CALL and RETURN edges that map between actual and formal parameters. In addition to edge types, CALL and RETURN edges also have edge labels, which are call site addresses. *Varied* analysis starts propagating from the independent variable node for `X`. When *varied* analysis finishes, all nodes in path X → F → G → B are marked as varied. Now, *useful* analysis starts from the dependent variable node `Y`. At this time, it immediately returns from the algorithm because node `Y` is not marked as varied. No variables are marked as active.

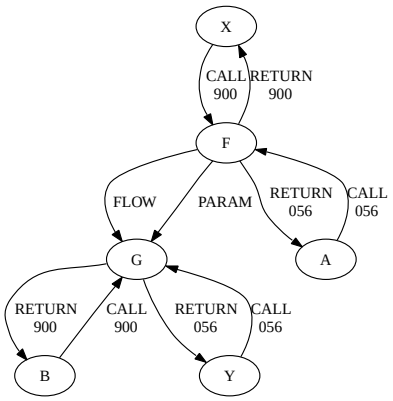

Figure 8. Variable dependence graph of Figure 4.

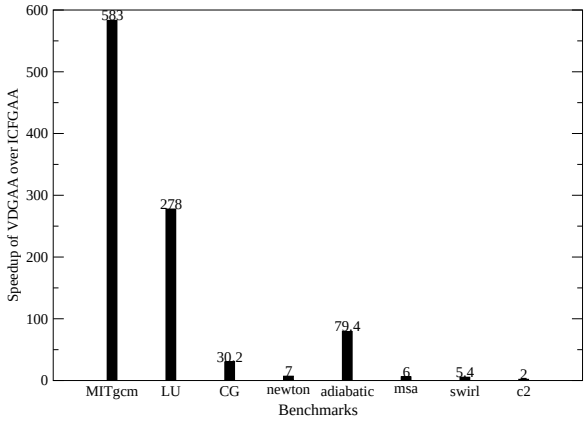

Figure 10. Speedups in analysis run time: VDGAA over ICFGAA.

4. Experiment

The VDGAA algorithm described in Section 3 is implemented in the OpenAnalysis environment. By defining intermediate representation (IR) interfaces for multiple compiler infrastructures, OpenAnalysis aims to make a single implementation of compiler analyses in IR independent fashion [18]. Although OpenAnalysis is used by multiple compiler infrastructures for multiple languages, for this experiment our implementation is linked into an automatic differentiation tool called OpenAD/F [19] in conjunction with the Open64/SL compiler infrastructure [15]. Figure 9 shows a block diagram of the experimental flow implemented as part of OpenAD/F, which is a source-to-source translator for Fortran. The machine we used has a 1.86 GHz Pentium M processor with 2 MB L2 cache and 1 GB DRAM memory.

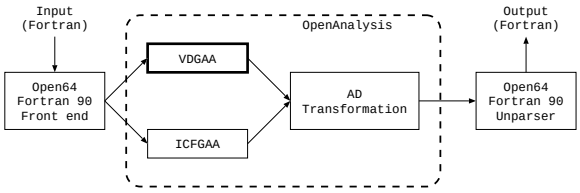

Figure 9. OpenAD automatic differentiation tool.

Table 3 shows the applications used in our experiments. *MITgcm* is a numerical model of the atmosphere, ocean, and climate [1]. While the code size is several hundred thousand lines, we use a stripped version. *LU* and *CG* are obtained from NAS parallel benchmarks [20]. Since the analyses do not understand the semantics of MPI calls, we augmented *LU* and *CG* with global variables and forced all communicated variables to be active by declaring them both independent and dependent. *Newton* implements Newton's method and Rosenbrock function. The *adiabatic* subroutine models adiabatic flow, a commonly used module in chemical engineering. *Msa* and *swirl* are from the MINPACK-2 test collection [3] and compute the minimal surface area and the swirling flow problem, respectively. C2 implements an ordinary differential equation solver.

Figure 10 shows the speedups of the new analysis over ICFGAA. The speedups range from 583 times for MITgcm to 2 times for C2. Notice the correlation between the speedups and the code sizes of Table 3. The speedup increases as the program size grows. *Newton* was an exception because, compared to its large code size, only the small Rosenbrock function was differentiated.

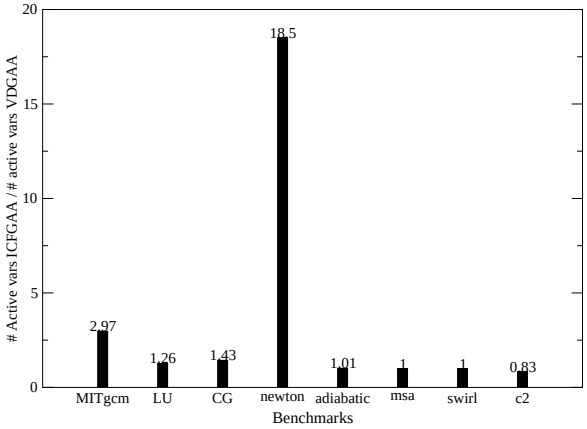

Figure 11. Reduction in number of active variables.

Our next interest is the quality of the produced outputs. While we can conservatively assume that all variables are active and still the generated codes will produce the correct numerical results, computing unnecessary derivatives for passive variables not only is a source of longer execution time but also requires more memory space possibly thrashing the disks for swapping. Figure 11 shows the reduction in number of active variables. The reduction is computed by dividing the number of active variables produced with ICFGAA by that of VDGAA. Computing the reduction factor this way is conservative because we do not know the exact number of active variables and assume that all found active variables are overestimations. In reality, however, a large portion of active variables identified by VDGAA are truly active, and the reduction factors are expected to be much higher. For *MIT-*

Table 3. Benchmarks.

Benchmarks	Description	Source	# lines
MITgcm	MIT General Circulation Model	MIT	13787
LU	Lower-upper symmetric Gauss-Seidel	NASPB	5951
CG	Conjugate gradient	NASPB	2480
newton	Newton's method + Rosenbrock function	ANL	2189
adiabatic	Adiabatic flow model in chemical engineering	CMU	1009
msa	Minimal surface area problem	MINPACK-2	461
swirl	Swirling flow problem	MINPACK-2	355
c2	Ordinary differential equation solver	ANL	64

gcm, ICFGAA produced 758 active variables, whereas VDGAA found only 255 among them. The reduction is not very large for *LU* and *CG* because of the global variable inserted to model MPI calls. For *newton*, ICFGAA overestimated a significant number of active variables. While the independent and dependent variables are declared within a small procedure, for the *varied* and *useful* analysis of ICFGAA, values are propagated along the call and return edges into the calling function, leading to the overestimation of active variables in the calling procedure. For *adiabatic*, VDGAA found two fewer active variables than did ICFGAA, but both analyses found the same set of active variables for *msa* and *swirl*. C2 is interesting because VDGAA overestimated one more active variable than did ICFGAA. The overestimation is caused by flow insensitivity of VDGAA where the statement order information is lost. For all other benchmarks shown in the graph, the sets of active variables found by VDGAA are the subsets of those found by ICFGAA. In other words, only C2 has one passive variable that is overestimated as active by VDGAA but correctly determined as passive by ICFGAA. However, we found some other applications where VDGAA overestimated more active variables than did ICFGAA. The most common case is shown in Figure 12. In (a), `a` is active because there is a value flow path from an independent variable `x` to a dependent variable `y` through `a`: $x \rightarrow f1 \rightarrow b1 \rightarrow a \rightarrow b1 \rightarrow g1 \rightarrow y$. In (b), `a` is not active because there is no such value flow path from `x` to `y` through `a`. The value of `a` returned from the first call to `foo2` is passed back to `b2` of the second call to `foo2` but is killed by the first assignment within `foo2`. Whereas ICFGAA correctly discovers this, VDGAA fails because there is no difference between the VDGs of `foo1` and `foo2`. In order to eliminate these overestimations caused by flow insensitivity of VDGAA, a global reaching definition analysis can be used.

```
subroutine head(x,y)

c$openad INDEPENDENT(x)
   call foo1(x, a, y)
   call foo1(x, a, y)
c$openad DEPENDENT(y)
end subroutine

subroutine foo1(f1,b1,g1)
   g1 = b1
   b1 = f1
end subroutine
```

(a) 'a' is active

```
subroutine head(x,y)

c$openad INDEPENDENT(x)
   call foo2(x, a, y)
   call foo2(x, a, y)
c$openad DEPENDENT(y)
end subroutine

subroutine foo2(f2,b2,g2)
   b2 = f2
   g2 = b2
end subroutine
```

(b) 'a' is passive

Figure 12. Overestimation of active variables in VDGAA.

This outstanding output quality of CSFI activity analysis with respect to that of CIFS activity analysis is beyond our initial expectation. Several factors can help explain this result. The first is the way programmers use variables. For scientific applications written in Fortran, usually variables are defined once, before being used. Second, by the nature of activity analysis, the analysis value for a variable is binary, either active or passive. Hence, even when a variable is defined and used multiple times, it is highly likely that a correct value still is determined for the variable. For example, consider a variable defined and used 10 times each. VDGAA determines the activity of the variable correctly if at least one of the following conditions is met: (1) at least one def-use pair is on a value flow path from an independent variable to a dependent variable, (2) no definition of the variable has a value flow path from any independent variable, or (3) no use of the variable leads to any dependent variable. VDGAA may overestimate the variable as active if there is a definition leading from an independent variable and a use leading to a dependent variable in VDG but the definition does not reach the use in the program because either the use textually precedes the definition or the def and use are in two mutually exclusive control paths. Thus, this comparison result between CSFI and CIFS activity analyses on scientific applications written in Fortran should not be blindly generalized to other analyses nor to other applications written in other programming languages. Lastly, ICFGAA makes some conservative assumptions. For example, any variable aliased with an active variable is assumed to be also active.

5. Related Work

Activity analysis has been known for some time [5], but only recently have algorithms been developed and described for the analysis. Hascoet et al. have described an algorithm based on iterative dataflow analysis framework [9]. Their data flow equations are similar to the ones used in ICFGAA. Kreaseck et al. investigated dynamic activity analysis that make decisions during run time as to computing derivative values [11]. Strout et al. used activity analysis as an example to verify their data flow analysis framework extended for MPI [7] programs [17]. Fagan and Carle described the static and dynamic activity analysis in ADIFOR 3.0 [6]. Similar to VDGAA, their static activity analysis is context-sensitive flow-insensitive. As in VDGAA, they apply transitive closure to compute transitive dependencies of all pairs of variables in a procedure.

Program slicing is a technique to find all statements and predicates of a program that might affect a variable *x* at a program point P [21]. Forward slicing is similar to *varied* analysis, and backward slicing is similar to *useful* analysis of VDGAA. *Program chopping* is a technique to obtain all program elements (statements or pred-

icates) that are used to convey effects from a source element *s* to a target element *t* [14]. Comparable to PARAM edges in VDGAA, they compute *summary edges* among actual parameters of procedure calls. Instead of computing transitive closure for all local variables, they restrict the computation to procedure parameters. Thus, the complexity to compute summary edges is $O(pn^2)$ as opposed to $O(n^3)$ for Warshall's algorithm, where p is the number of procedure parameters and n is the number of variables used in the procedure [13].

Program chopping returns program elements such as statements and predicates as an output whereas activity analysis returns active variables. It is conceivable to use program chopping to find activity of variables by ignoring the predicates in the output and taking the variables being defined by the statements as active variables. However, the active variables obtained this way may include the variables that depend on independent variables through control dependence and the ones on which dependent variables depend through control dependence. Consequently the output can be more conservative than that of VDGAA. Compared with VDGAA, program chopping has a wider range of applications. However, VDGAA is simpler and cheaper than program chopping. For example, in order to build a system dependence graph for program chopping, data flow analysis is performed to compute the set of reaching definitions. For VDGAA, building a variable dependence graph requires a single scan of the input program. Although we use an $O(n^3)$ algorithm to generate PARAM edges, this is only for simplicity, and the $O(pn^2)$ algorithm of program chopping can replace Warshall's algorithm in VDGAA. Furthermore, the number of parameters (represented as p) of VDGAA can be much smaller than that of program chopping because, in program chopping, global variables are treated as extra parameters. In addition, the cost of graph navigation algorithm for VDGAA is $O(P \times E + CSites \times Params^2)$ whereas it is $O(Params \times (P \times E + CSites \times Params^2))$ [1] for program chopping where, P is the number of procedures in the program, E is the maximum number of edges for the graph of any procedure, $CSites$ is the total number of call sites in the program, and $Params$ is the maximum number of procedure parameters in any procedure.

6. Conclusion

Activity analysis is essential for automatic differentiation tools by allowing them to generate efficient derivative codes that run faster with less memory. In this paper, we describe a novel context-sensitive flow-insensitive activity analysis, called VDGAA, and provide a comparison with an existing context-insensitive flow-sensitive activity analysis, called ICFGAA. In our experiments on eight benchmarks, the speedups of VDGAA over ICFGAA range from 2 to 583 times. For most benchmarks, VDGAA overestimates the same or fewer active variables than ICFGAA.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We thank Gail Pieper for proofreading several revisions.

[1] Nontruncated, same-level chopping

References

[1] http://mitgcm.org/.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Brett M. Averick, Richard G. Garter, and Jorge J. More. MINPACK-2 Test Problem Collection. Technical Memorandum ANL/MCS-TM-150, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, May 1991.

[4] Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA, 1996.

[5] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[6] Mike Fagan and Alan Carle. Activity Analysis in ADIFOR: Algorithms and Effectiveness. Technical Report TR04-21, Department of Computational and Applied Mathematics, Rice University, Houston, TX, November 2004.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[8] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, PA, 1991.

[9] L. Hascoët, U. Naumann, and V. Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.

[10] Uday P. Khedker and Dhananjay M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, 1994.

[11] Barbara Kreaseck, Luis Ramos, Scott Easterday, Michelle Strout, and Paul Hovland. Hybrid static/dynamic activity analysis. In *Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA'04)*, Reading, England, 2006.

[12] Arun Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering*, pages 35–44, Baltimore, MD, 1993.

[13] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20, 1994.

[14] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 41–52, 1995.

[15] Rice University. Open64 project. http://www.hipersoft.rice.edu/open64/.

[16] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *International Symposium on Static Analysis*, pages 16–34, 1997. Lecture Notes in Computer Science, Vol. 1302, Pascal Van Hentenryck (ed.), Springer-Verlag, New York, NY.

[17] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-flow analysis for MPI programs. In *International Conference on Parallel Processing*, 2006.

[18] Michelle Mills Strout, John Mellor-Crummey, and Paul D. Hovland. Representation-independent program analysis. In *Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[19] Jean Utke. OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[20] Rob F. van der Wijngaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Advanced Supercomputing (NAS) Division, October 2002.

[21] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.